Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS EU 2024 - Selected Papers from the 11th Annual Digital Forensics Research Conference Europe

Forensic implications of stacked file systems

Jan-Niclas Hilgert [*], Martin Lambertz, Daniel Baier

Fraunhofer FKIE, Zanderstr. 5, 53177 Bonn, Germany

A R T I C L E I N F O

Keywords:
Storage forensics
File systems
Stacked file systems
Distributed file systems
moosefs
Glusterfs
Ecryptfs

A B S T R A C T

While file system analysis is a cornerstone of forensic investigations and has been extensively studied, certain file system classes have not yet been thoroughly examined from a forensic perspective. Stacked file systems, which use an underlying file system for data storage instead of a volume, are a prominent example. With the growth of cloud infrastructure and big data, it is increasingly likely that investigators will encounter distributed stacked file systems, such as MooseFS and the Hadoop File System, that employ this architecture. However, current standard models and tools for file system analysis fall short of addressing the complexities of stacked file systems. This paper highlights the forensic challenges and implications associated with stacked file systems, discussing their unique characteristics in the context of forensic analyses. We provide insights through three analyses of different stacked file systems, illustrating their operational details and emphasizing the necessity of understanding this file system category during forensic investigations. For this purpose, we present general considerations that must be made when dealing with the analysis of stacked file systems.

1. Introduction

File system analysis is undeniably an essential part during any digital forensic investigation involving storage devices. Its goal is to identify and extract files and their corresponding metadata including deleted information. Brian Carrier already laid a profound foundation for this research area almost 20 years ago covering various file systems, some of which are still being used today such as FAT, NTFS and Ext (Carrier, 2005). According to his model for file system forensic analysis, *traditional file systems* store their data on a volume, e.g. a partition or a RAID. Since these volumes are transparent to the file system itself, the underlying implementation creating the volume is responsible for the final transformation and distribution of the actual data. Analyzing these volumes is completely detached from the actual file system at hand and can thus be first addressed in the volume analysis phase, which is then followed by the final file system analysis.

As pointed out by Hilgert et al., these two phases become intertwined, requiring an additional layer in the model when dealing with *pooled file systems* (Hilgert et al., 2017). These file systems utilize multiple disks for redundancy or performance but do not require any extra soft- or hardware for this purpose. In these cases, the file systems themselves handle the distribution of the data across the underlying layer, i.e. volume. Still, the file systems presented in their work stored their data directly on the underlying volume layer.

This work takes a closer look at the forensic analysis of *stacked file systems*. These file systems might also handle the distribution of their data, but they are distinctively characterized by their method of data storage: they do not store their data on a volume or disk but rather on another file system creating new opportunities and challenges for forensic analysis practitioners encountering these file systems. Given the adoption of this concept in distributed file systems like MooseFS and the Hadoop File System, equipping forensic analysts with the knowledge to handle these systems during investigations proficiently is essential.

For this purpose, this paper discusses crucial aspects of the analysis of stacked file systems. To accommodate this, we have revised the standard workflow for file system forensic analysis, making it suitable for the intricacies of stacked systems. We also describe a core set of forensic implications for analyzing stacked file systems, complemented by illustrative findings from three different file systems. The knowledge gathered from our experiments emphasizes the necessity of understanding these implications and is a vital reference for forensic analysts.

2. Stacked file systems

Stacked or *stackable file systems* store their data on another file system, including both data and metadata, which might be stored in a

* Corresponding author.
E-mail addresses: jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), martin.lambertz@fkie.fraunhofer.de (M. Lambertz), daniel.baier@fkie.fraunhofer.de (D. Baier).

https://doi.org/10.1016/j.fsidi.2023.301678

specialized file format. We denote the stacked file system as the *upper file system* and its files as the *upper files*, which are the files accessible when the file system is mounted. The underlying file system it relies on is termed the *lower file system* storing the *lower files*, as depicted in Fig. 1.

In instances where the upper and lower file systems operate on the same machine, the stacked file system is termed as *local*. Nevertheless, an upper file can encompass multiple lower files, potentially distributed across various detached lower file systems. Given this, the concept of stacked file systems is frequently employed within *distributed* stacked file systems like the Hadoop Distributed File System or MooseFS, as they can be constructed atop a pre-existing and reliable lower file system. Furthermore, distributed stacked file systems can be categorized as either *managed* or *unmanaged*. In a managed setup, a designated entity like a main daemon can be used to orchestrate tasks such as data distribution and managing the metadata of the upper file system. Conversely, in an unmanaged configuration, the systems housing the lower file systems inherently possess all the requisite data to construct the upper file system. Both of these types can be encountered during forensic investigations due to the increasing usage of distributed storage in cloud environments. Hence, comprehending the forensic implications and nuances of stacked file system analysis is crucial.

2.1. Related work

A detailed concept of stacking file system layers was already presented in 1994 (Heidemann and Popek, 1994). However, this work focuses on file system development and describes stacking as a method to leverage already existing file systems facilitating the development process of new file systems and features. A few years later, Erez Zadok utilized the concept of stacked file systems to implement a wrapper file system called Wrapfs (Zadok, 1999). While it still stores its data on a lower file system, Wrapfs can be used to create arbitrary upper file systems, for example to provide encryption or prevent deletions of files. In 2007, Zadok together with others discussed various issues of stacked file systems within Linux, such as cache coherency between the upper and lower file system (Sipek et al., 2007). Furthermore, file systems for secure deletion and tracing of file interactions based on the concept of stacked file systems have been proposed (Bhat and Quadri, 2012; Aranya et al., 2004).

While all of the aforementioned research does focus on stacked file systems, it does not cover them from a forensic point of view. Still, limited research on the forensic analysis of distributed stacked file systems has been published (Asim et al., 2019; Harshany et al., 2020). take a closer look at the Hadoop Distributed File System. While their work yields interesting results, such as analyzing various commands and reconstructing distributed data, they do not address the underlying file system used by HDFS. Another analysis of a distributed stacked file system was performed in (Martini and Choo, 2014). During their analysis of XtreemFS, the authors also focused on the Object Storage Devices storing the lower file systems including its identification. However, their work falls short in providing a detailed discussion of general implications of the underlying concept of stacked file systems. Furthermore, the additional value of an analysis of the underlying file system is not examined.

2.2. Forensic analysis of stacked file systems

In addition to the research gap, it is important to note that this deficiency extends to forensic tools as well. Current tools, like The Sleuth Kit, are equipped to analyze various lower file system types but lack the functionality to associate them with any existing upper file system. This limitation underscores the need for an updated standard workflow for file system forensic analysis, as depicted in Fig. 2, to effectively handle the complexities of stacked file systems.

In the revised workflow, the initial steps shown in white remain, but we introduce an additional phase, highlighted in purple, specifically dedicated to the analysis of stacked file systems, building upon the results from the prior analysis of the lower file system. This emphasizes that the detection and analysis of traditional file systems continue to be the foundational elements of the process. However, these steps may now yield multiple lower files or metadata files associated with stacked file systems, requiring thorough examination in the newly added step to ensure a comprehensive forensic analysis. Crucially, the results from the stacked file system analysis must also be correlated with information derived from the lower file system analysis and vice versa.

The remainder of this paper deals with the additional step of stacked file system forensics and integration into forensic investigations. In particular, we look at six specifics, we believe are essential for file system analysis: 1) Identification of Stacked File Systems, 2) Correlation of File Names, 3) Data Reconstruction, 4) Timestamps and their Update Behavior, 5) Slack Space and 6) Possibilites for File Recovery.

2.3. Experimental setup

To derive the most comprehensive guidance possible, it is crucial to include a diverse range of stacked file systems in the experiments.

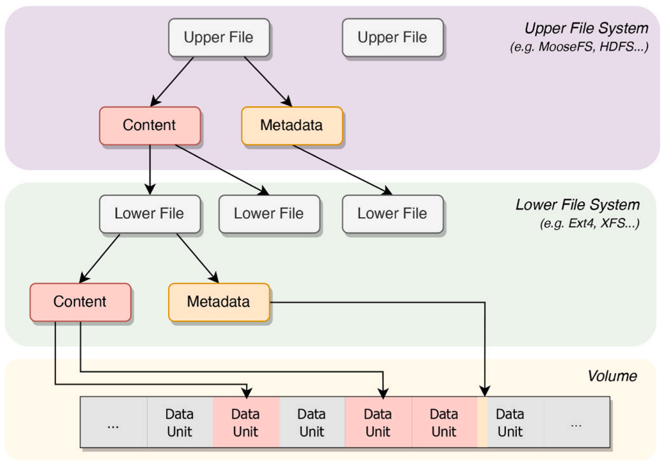


Fig. 1. Overview of a stacked file system utilizing a traditional lower file system for data storage.

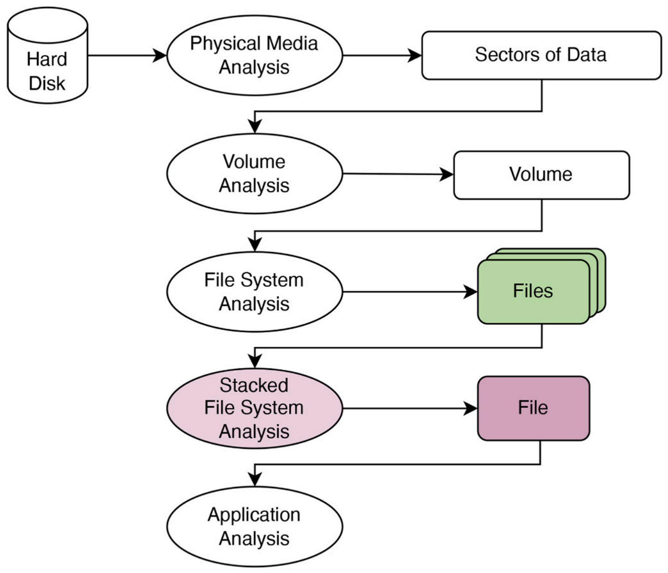


Fig. 2. Extension of Brian Carrier's model for the applicability of stacked file systems.

Accordingly, three distinct stacked file systems, previously overlooked in research, were selected as representative examples:

MooseFS released in 2008, is an open-source, *managed, distributed stacked file system* designed for big data storage. Its architecture includes Chunk Servers that store data, a Master Server managing metadata, Metaloggers for metadata backup, and a client interface for mounting the file system. In MooseFS, large files are split into smaller chunks distributed across multiple servers.

GlusterFS is an *unmanaged, distributed stacked file system that*differs from MooseFS in that it lacks a dedicated master server. Instead, its storage servers form a *trusted pool* by connecting directly to each other. It supports any file system as a *brick*, the lower file system for storing data. These bricks are combined to create a *volume*, which is subsequently mounted by a client.

eCryptfs introduced in 2005 as a cryptographic file system to operate on top of an existing file system (Halcrow and ecryptfs, 2005), was integrated into the Linux kernel in version 2.6.19. Although superseded by other mechanisms such as LUKS, eCryptfs remains a notable early example of stacked file systems. It functions as a *local stacked file system*, not used in a distributed manner, and is mounted by specifying a source directory from the lower file system to store its data.

This variety ensures a thorough exploration of the potential scenarios forensic experts may encounter. For our experiments, the stacked file systems were setup, mounted and populated with arbitrary data. Specifics of each experiment are presented in the corresponding section. As a lower file system during the experiments, we utilized Ext4 due to its widespread use and to keep the results comparable. Drawing on these findings, the following sections also outline practical key takeaways to aid forensic investigators in their work with stacked file systems.

3. Identification of stacked file systems

As described in Section 2.3, it is crucial to identify a stacked file system following the analysis of the lower file system. During these experiments, the lower file systems were analyzed for any indicators hinting at the usage of a stacked file system.

3.1. Findings

3.1.1. MooseFS

As soon as a file system is being used as part of a chunk server in MooseFS, a distinct hierarchy of directories from `00` up to `FF` is created on it. These directories are used to store the chunks, which in turn utilize a file name pattern like `chunk_0000000000000001_00000001.mfs` consisting of an identifier, the chunk ID, a corresponding version and the file extension. Lower files in MooseFS can also be identified by their internal structure that can be inferred by taking a look at the open source code of the file system. In the default MooseFS installation, i.e. not the light version, each chunk begins with a `0x2000` bytes long header. It starts with either a signature of `MFSC 1.0` or `MFSC 1.1`, followed by eight and respectively four bytes representing the chunk ID and version, both of which can also found within the chunk's file name.

3.1.2. GlusterFS

A similar behavior can be observed on servers of a GlusterFS pool, when a volume is created and started. This includes a hidden `.glusterfs` directory storing directories named `00` up to `ff`. Each upper file in GlusterFS is assigned a UUID referred to as the *GlusterFS internal file identifier (GFID)*. This GFID names each lower file inside the hidden hierarchy. GlusterFS also mirrors the upper system's structure in the lower system using hard links as depicted in Fig. 4. While GlusterFS does not make use of any specific internal structure within its lower files, it uses extended attributes to store meta information about its files.

3.1.3. eCryptfs

eCryptfs on the other hand does not create a unique hierarchy on the lower file system. Instead, the hierarchy of the files and directories of the upper file system are stored in an identical way on the lower file system. If file name encryption is enabled, the distinct prefix `ECRYPTFS_FNEK_ENCRYPTED` defined in the Linux kernel source is used for each lower file. Lower files in eCryptfs contain magic markers stored in a special header. These markers can be detected by performing an XOR operation on bytes 9–12 of the file at hand using the magic `0x3c81b7f5`. The resulting 8 bytes should match bytes 13–16 in case of an eCryptfs lower file.

3.2. Key takeaways

Depending on the stacked file system at hand various types of indicators resulting from the analysis of the lower file system can be used for its identification. This includes distinct hierarchies, file structures as well as certain extended attributes. Furthermore, the internal structure of lower files can be used to identify them directly, for example in cases in which they are included in a backup outside of the lower file system.

Once identified, investigators can mount the stacked file system using its native software or conduct an in-depth forensic examination. However the current shortfall in forensic tools specifically designed for stacked file system analysis necessitates manual reconstruction of the system under investigation at the moment.

4. Correlation of file names

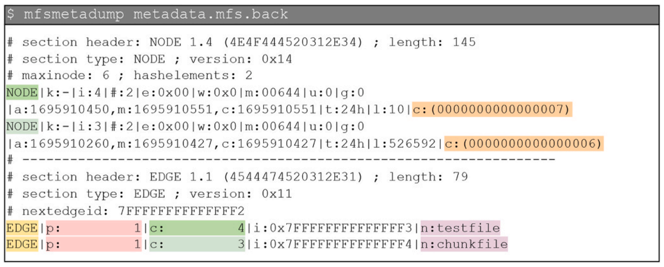
For a more comprehensive analysis and deeper understanding, it is essential to establish the relation between the names of upper files and the corresponding lower files that represent them. During this experiment, we analyzed if and how this connection could be determined. Furthermore, we examined how the entire hierarchical structure of the upper file system is reflected within the lower file system.

4.1. Findings

4.1.1. MooseFS

In MooseFS, neither the header, the file name or any other metadata of a lower file contain any reference to the original upper file. In order to obtain this relation and thus also the file name, it is necessary to analyze information stored on the Master Server. By default, chunk servers use the DNS name `mfsmaster` to connect to the Master Server. However, this can be configured within the chunk server's configuration stored in`/etc/mfs/mfschunkserver.cfg`. The Master Server stores its metadata files within the directory`/var/lib/mfs`, including `metadata.mfs.back`. This metadata file can be extracted and subsequently inspected or analyzed using the `mfsmetadump` tool. During our experiments, recent MooseFS updates were not instantly reflected in the metadata file, requiring a Master Server restart to save these changes.

Fig. 3 illustrates the `mfsmetadump` utility output. In MooseFS, file names are stored as `EDGE`s in the filesystem tree, found in the `EDGE` section. Each line represents a file, detailing the parent inode, child inode, and file's name. The child inode number can be used to link an



```
$ mfsmetadump metadata.mfs.back

# section header: NODE 1.4 (4E4F444520312E34) ; length: 145
# section type: NODE ; version: 0x14
# maxinode: 6 ; hashelements: 2
NODE|k:-|i:4|#:2|e:0x00|w:0x0|m:00644|u:0|g:0
|a:1695910450,m:1695910551,c:1695910551|t:24h|l:10|c:(0000000000000007)
NODE|k:-|i:3|#:2|e:0x00|w:0x0|m:00644|u:0|g:0
|a:1695910260,m:1695910427,c:1695910427|t:24h|l:526592|c:(0000000000000006)
# ------------------------------------------------------------------
# section header: EDGE 1.1 (4544474520312E31) ; length: 79
# section type: EDGE ; version: 0x11
# nextedgeid: 7FFFFFFFFFFFFFF2
EDGE|p:           1|c:           4|i:0x7FFFFFFFFFFFFFF3|n:testfile
EDGE|p:           1|c:           3|i:0x7FFFFFFFFFFFFFF4|n:chunkfile
```

Fig. 3. Excerpt of the `mfsmetadump` tool displaying metadata of a MooseFS file system.

entry to a corresponding NODE section entry, which represents an upper file.

4.1.2. *GlusterFS*

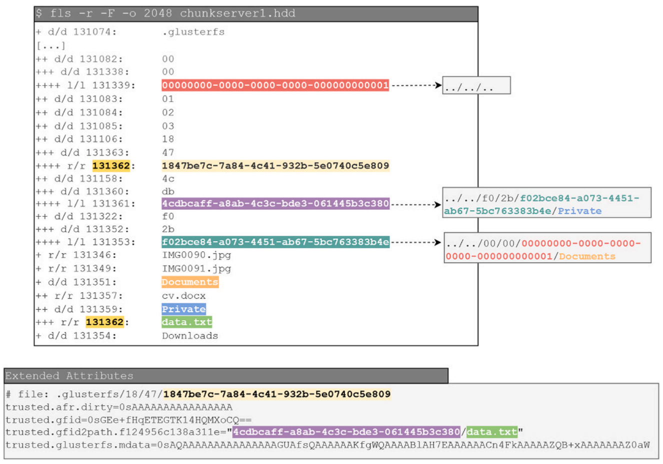
Fig. 4, the lower files 1847be7c-7a84-4c41-932b-5e0740c5e809 and data.txt share the same inode number. Additionally, GlusterFS also utilizes extended attributes and soft links, which can be analyzed to infer the hierarchy. The extended attributes of the lower file contain a reference including the original file's name as well as the GFID of the directory, in which it was stored. The lower file belonging to this directory is again a soft link pointing to its own parent directory and so on.

4.1.3. *eCryptfs*

If the eCryptfs file system is mounted files within the upper file system can be matched to the files in the lower file system by comparing the corresponding inode numbers. This is already implemented in the ecryptfs-find utility. By default, the file names of the lower files are identical to the file names of the corresponding upper files. In case of file name encryption, eCryptfs utilizes a file name encryption key (FNEK), which is required to reveal the original file name of the lower file at hand. However, eCryptfs stores a hex signature of the utilized FNEK within all of the encrypted files names. For this reason, it is still possible to infer, which lower files were encrypted using the same FNEK and thus probably belonged to the same mounted file system. The signature of the FNEK is encoded within the FNEK-encrypted file name, also referred to as a Tag 70 packet and follows the packet type 0x46 and the length of the packet. By decoding the file name it is possible to extract the signature of the FNEK used, which can be used for further analyses.

4.2. *Key takeaways*

Our experiments indicate that in local or unmanaged distributed stacked file systems, it is generally possible to deduce the original file names and file system hierarchy. This is to be expected as for these kinds, the corresponding metadata can be found within the lower file system. In contrast, with stacked file systems that incorporate a management component, e.g. a dedicated server, it becomes vital to identify and extract the metadata that holds this information. Our findings demonstrate how this analysis can be executed for stacked file systems like MooseFS, enabling the determination of the relationship between upper and lower files. However, this task varies significantly depending on the specifics of the stacked file system, necessitating customized implementations within forensic tools.



```
$ fls -r -F -o 2048 chunkserver1.hdd
+ d/d 131074:        .glusterfs
[...]
++ d/d 131082:       00
+++ d/d 131338:      00
++++ l/l 131339:     00000000-0000-0000-0000-000000000001 --> ../../..
++ d/d 131083:       01
++ d/d 131084:       02
++ d/d 131085:       03
++ d/d 131106:       18
+++ d/d 131363:      47
++++ r/r 131362:     1847be7c-7a84-4c41-932b-5e0740c5e809
++ d/d 131158:       4c
+++ d/d 131360:      db
++++ l/l 131361:     4cdbcaff-a8ab-4c3c-bde3-061445b3c380 --> ../../f0/2b/f02bce84-a073-4451-ab67-5bc763383b4e/Private
++ d/d 131322:       f0
+++ d/d 131352:      2b
++++ l/l 131353:     f02bce84-a073-4451-ab67-5bc763383b4e --> ../../00/00/00000000-0000-0000-0000-000000000001/Documents
+ r/r 131346:        IMG0090.jpg
+ r/r 131349:        IMG0091.jpg
+ d/d 131351:        Documents
++ r/r 131357:       cv.docx
++ d/d 131359:       Private
+++ r/r 131362:      data.txt
+ d/d 131354:        Downloads
```

```
Extended Attributes
# file: .glusterfs/18/47/1847be7c-7a84-4c41-932b-5e0740c5e809
trusted.afr.dirty=0sAAAAAAAAAAAAAAAA
trusted.gfid=0sGEe+fHqETEGTK14HQMXoCQ==
trusted.gfid2path.f124956c138a311e="4cdbcaff-a8ab-4c3c-bde3-061445b3c380/data.txt"
trusted.glusterfs.mdata=0sAQAAAAAAAAAAAAAAAGUAfsQAAAAAAKfgWQAAAABlAH7EAAAAAACn4FkAAAAAZQB+xAAAAAAAZ0aW
```

Fig. 4. Example of a hierarchy on a lower file system in GlusterFS.

5. Data reconstruction

For the reconstruction of upper files from their corresponding lower files, analysts need to tackle common problems such as fragmentation and data transformation.

5.1. *Fragmentation*

In most cases, the content of a file does not fit into a single data unit, which is why file systems allocate multiple data units. While different allocation strategies may be used, it often results in file fragmentation. Thus, traditional file systems need to keep track of the exact data units used by a file as well as the order in which they belong. This fragmentation not only complicates forensic efforts but has also been a long-standing focal point of research (Garfinkel, 2007; van der Meer et al., 2021). Yet, the topic of fragmentation in stacked file systems has not been explored. To address this, we have created multiple large-sized upper files, aiming to analyze and understand the fragmentation patterns in the stacked file systems under study.

5.1.1. *Findings*

5.1.1.1. MooseFS. Our experiments demonstrated that MooseFS splits files larger than 64 MiB into multiple lower files, irrespective of the number of chunk servers. Although mountable with a single chunk server, MooseFS ideally operates with multiple, and it is advised to use at least three, as done in our experiments. By default, each chunk is replicated onto two of the three available chunk servers. Consequently, large files, fragmented into multiple chunks, may be distributed across all chunk servers within the MooseFS file system. Since information within the chunks themselves did not suffice to reassemble an upper file, it is necessary to consult the Master Server metadata to efficiently assemble fragmented upper files. As depicted in Fig. 3, the NODE section stores a list of chunks composing the upper file, each identified by a unique ID, which is also reflected in the chunk name on the lower file systems. Since it is unique to each chunk, it can also be used to identify replicas of chunks across multiple chunk servers.

5.1.1.2. GlusterFS. Depending on the type of volume used, fragmentation as well as replicas of lower files can be encountered. The most important volume types are:

- **Distributed:** In this default mode, upper files are not fragmented, but stored randomly across all available bricks, i.e. all available lower file systems.
- **Replica:** This mode is used to ensure redundancy by storing unfragmented upper files across multiple bricks similar to RAID mirrors. The corresponding lower files stored across multiple lower file systems can be correlated by their GFID file name.
- **Dispersed:** A dispersed volume can be compared to a RAID5-like volume. Data is split and stored across multiple lower file systems along with parity information. Again, fragments belonging together can be matched by their GFID. When creating a dispersed volume, it is possible to configure the number of bricks used for redundancy, i.e. how many bricks can be lost without causing any data loss.

The first two modes do not cause any fragmentation of upper files. However, the replicated mode causes multiple copies of the same files to be stored on multiple lower file systems, i.e. storage servers. To identify these lower file systems for further analysis, the configuration of the GlusterFS at hand can be utilized. A directory for each volume of a storage server can be found in its /var/lib/glusterd/vols/ directory. It stores the volume configuration in an info file and consists of a bricks subdirectory that offers configurations for each associated brick. These files appear across all storage servers in the GlusterFS pool that created the volume, detailing the server's hostname and brick path. Notably, the values `listen-port` and `brick-fsid` only seem to exist in the brick configuration of the respective server. This also allows for pinpointing the exact GlusterFS server at hand.

When dealing with dispersed volumes, upper files become fragmented within GlusterFS. An efficient way to identify a lower file of a dispersed volume is to analyze its extended attributes. Each chunk belonging to a dispersed file utilizes extended attributes in the `trusted.ec` name space, e.g. `trusted.ec.size` which stores the real size of the corresponding file. However, they do not contain any information about the order in which they should be reassembled. Furthermore, GlusterFS uses Erasure coding for dispersed volumes, which requires an additional step to obtain the original version of the file described in the next section.

5.1.1.3. eCryptfs. In eCryptfs upper files are not split into multiple lower files and thus no fragmentation occurs.

5.1.2. Key takeaways

Though our findings illustrated that fragmentation may not be as complex as with traditional file systems, it still has to be considered especially when dealing with distributed stacked file systems spanning across multiple lower file systems. In these cases, it is crucial to identify, which other servers were part of the stacked file system at hand in order to adequately extend the acquisition process.

Furthermore, our experiments showed that the upper file system's metadata plays a crucial role for an efficient reassembly of fragmented files highlighting the importance of dedicated approaches for stacked file system analysis. In absence of this information, correlating lower files via their timestamps is an alternative though less reliable due to discrepancies as discussed in Section 6.

5.2. Transformation

Unlike early file systems, more advanced ones like APFS or ZFS started to implement features such as encryption or compression. This resulted in some kind of transformation between a file's original content and the content stored on disk. A similar concept may also be employed by stacked file systems for various purposes like encryption or the utilization of erasure coding, when data is distributed. For this analysis, we compared the content of lower files to the original content stored within the upper files of the stacked file system.

5.2.1. Findings

5.2.1.1. MooseFS. Besides the inclusion of an extra `0x2000` byte chunk header, the open-source MooseFS leaves the original data unaltered.

5.2.1.2. GlusterFS. In distributed and replicated volumes, GlusterFS leaves the original content in lower files unchanged as well. However, for dispersed volumes, it employs a Reed-Solomon based Erasure coding. For an efficient recovery of dispersed files, we recreated the relevant GlusterFS setup to tackle the fragmentation as well as transformation hurdle. After the original GlusterFS configuration is identified as described in the previous section, it is possible to recreate a new GlusterFS volume using identical parameters. Afterwards, the obtained lower file systems can be copied to the freshly created GlusterFS bricks. It is essential to preserve extended attributes; failure to do so will lead GlusterFS to misidentify dispersed files. Additionally, the sequence of declaring bricks is crucial, i.e. the original first lower file system's data should populate the first brick in the new volume and so on. Any inconsistency led to reconstruction failure in GlusterFS in our tests.

5.2.1.3. eCryptfs. Since eCryptfs's main feature is encryption, file contents found on the lower file system are naturally encrypted. Additionally, the cryptographic context for each file is stored in a header preceding the encrypted data. The minimum size for this header is defined as 8192 bytes, resulting in slightly larger files on the lower file system compared to the original stacked file system. Though 8192 bytes is only the minimum size, we did not encounter any larger header sizes in our experiments including files up to 1 GB. The size of the original file is not encrypted and can be found in bytes 0–7 of the header, which starts directly at offset 0 of the lower file. Further information within the header includes the version as well as the encrypted session key used for the encryption of the file's content.

5.2.2. Key takeaways

Depending on the stacked file system at hand, practitioners can benefit from the absence of a transformation layer during their analysis. This enables them to analyze a the files of a lower file system without the need to perform an analysis of the upper stacked file system. However, in certain cases, when encryption or error encoding is utilized, it is required to retranslate the content of lower files to obtain the original file content. We have illustrated various considerations that have to be made when using native software to perform this task for GlusterFS. Yet again, this strongly depends on the features of the stacked file system at hand.

6. Timestamps and their update behavior

In traditional file systems, timestamps are stored along the metadata of the files and include information about the last Access, Modification, Change and in some cases Birth time of a file. The intricacies and challenges of interpreting timestamps are well-recognized within the digital forensic community (Raghavan, 2013).

Naturally, these timestamps retain their critical importance in the context of stacked file systems. However, we encounter an additional layer of timestamp sources:

- **Upper file system:** Timestamps of the upper file system refer to the upper files and consist of one set of timestamps per upper file. The way this meta information is stored is completely specific to the upper file system itself.
- **Lower file system:** Employing an additional file system to store file content introduces an extra layer of timestamps stored along the lower files in the lower file systems.

Moreover, it is equally important to grasp the timestamp update behavior within both the upper and lower file systems as well as how they affect each other. In our experiments, we conducted fundamental file operations like creating and modifying files to examine how the stacked file systems in question update timestamps. This investigation encompassed both the lower and upper file systems, with a particular focus on understanding how timestamps in the latter could be accurately retrieved. We kept a multi-server configuration for the distributed file systems to observer the timestamp update behavior across multiple lower file systems.

6.1. Findings

The initial part of this section details the findings on timestamp sources, while the subsequent sections explore the timestamp update behavior of the corresponding file system.

6.1.1. Timestamp sources

MooseFS keeps track of the timestamps for all of its upper files within the metadata that can be found on the Master Server or Metaloggers. This information can be extracted by using the `mfsmetadump` utility as shown previously in Fig. 3. In GlusterFS, this information is not stored in an external file, but directly within the extended `trusted.glusterfs.mdata` attribute of the corresponding lower files across all bricks. Any change to the upper file's timestamps inevitably results in an update of the metadata of the lower files. The actual timestamps can be extracted from the decoded Base64 string stored within the extended attribute. It holds 8 byte timestamps in seconds followed by the timestamp for nanoseconds following the big-endian format as shown in Fig. 5 eCryptfs relies solely on the timestamps already present in the lower file system, without storing any additional timestamp information.

6.1.2. Update behavior for MooseFS

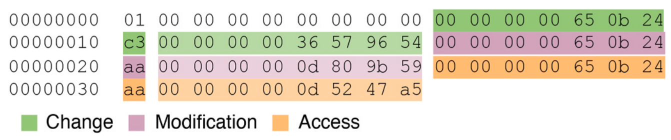
When a file smaller than the maximum chunk size is created, two identical chunk copies are made on two out of three chunk servers. Although MooseFS sets the Modification and Change timestamps of the upper file identically, the Access timestamp appeared slightly earlier in our tests. This pattern was also seen in timestamps of the corresponding chunk servers. Notably, the birth timestamp from the lower file system is not reflected in MooseFS. Furthermore, it was observed that different chunk servers displayed varying timestamps for the same chunk.

When an upper file is modified, its Modification and Change timestamps are updated to the same value. The same holds true for the corresponding chunks stored within the lower file systems. However, timestamps might again vary across chunk servers. If the upper file's timestamps are changed without data alteration, e.g. by utilizing the `touch` command in Linux, the chunk timestamps remain unaffected.

The update of File Access timestamps is rather complex and depends on multiple of factors:

- **MooseFS Configuration:** The `ATIME_MODE` in the Master Server config determines the Access time update policy for upper files. Default is always, with options like ”always for files” or ”never” (similar to Linux's `noatime`).
- **Client Caching:** When mounting MooseFS, it is possible to set a *data cache mode.* Options include `DIRECT` (no caching) and `YES` (always use cache). Default is `AUTO`, which behaved like `YES` in our tests.
- **Chunk Pre-Fetch:** For performance, MooseFS uses pre-fetch and read-ahead algorithms on chunk servers to pre-load expected chunks into the OS memory. This is hardcoded and cannot be changed.
- **Lower File System Configuration:** The lower file system on the chunk server has its own Access timestamp policy. In Linux, the default is `relatime`, which doesn't update Access times with every access.

In MooseFS, the file access timestamps for upper files are influenced by its configuration and client caching. It was observed that when *client caching is disabled*, every file access updates the Access timestamps on the client, which the Master Server adopts in the default configuration. If *client caching is enabled* however, the Access timestamp stamp of a file is only updated on its first access or when it gets reloaded into cache. If MooseFS is however configured to never Modification the Access timestamps, client-side updates aren't stored on the Master Server and are lost almost instantaneously. In our tests, Access timestamps for lower files were updated upon the chunk server daemon's initial start, provided its access time mode was set accordingly, e.g. using the `strictatime` option. With 10,000 files (and corresponding lower files), Access timestamps changed post-daemon start without client read requests. This is likely due to MooseFS's pre-fetch algorithms reading data in memory for some time, though no clear order was discernible.



```
00000000  01 00 00 00 00 00 00 00 00 00 00 00 00 65 0b 24
00000010  c3 00 00 00 00 36 57 96 54 00 00 00 00 65 0b 24
00000020  aa 00 00 00 00 0d 80 9b 59 00 00 00 00 65 0b 24
00000030  aa 00 00 00 00 0d 52 47 a5
```

🟩 Change 🟪 Modification 🟧 Access

Fig. 5. Structure of the `trusted.glusterfs.mdata` extended attribute containing timestamps in GlusterFS.

Reading all files from the client (with caching off and default MooseFS settings) resulted in the updating of all 10,000 Access timestamps in the lower file system. Yet, in setups with fewer upper files, Access timestamps of chunks only updated during daemon startup, not during later client requests. The cause for this disparity is still unclear and requires further research. Given these complexities, interpreting Access timestamps on MooseFS's lower file systems demands caution.

On the other hand, Modification timestamps were consistently accurate and updated as anticipated, which is especially relevant for large upper files generating multiple chunks.

Large files: In MooseFS, files exceeding the maximum chunk size are divided into multiple chunks. When such a large file is created, the timestamps in MooseFS and the underlying file system are set in the manner previously detailed. Thus, a 200 MB upper file results in eight (four distinct but replicated) lower files, each with unique timestamps, spread across three chunk servers.

In our experiment, we modified the first bytes of the 200 MB file. While MooseFS only holds a singular set of timestamps for the upper file, the Modification and Change timestamps are updated the same way regardless of the position, in which the file is modified. On the lower file systems however, only the Modification and Change timestamps of the impacted chunk, the first of four, were updated across the two chunk servers hosting that chunk. A similar pattern was observed when other sections of the file were altered: only the relevant chunk's timestamps changed. This level of granularity provides a more intricate view into file modifications on the upper file system.

In our MooseFS experiments, we observed an unexpected behavior where chunks sometimes moved between chunk servers after file modification or idling periods. While MooseFS naturally rebalances chunks across servers, the reasons for these specific movements were unclear. Crucially, this behavior has implications for timestamps. When a chunk is relocated to a new server, it behaves as if it's newly created, thus resetting all its timestamps to the time of the relocation.

6.1.3. Update behavior for GlusterFS

When a file is created in GlusterFS, the Modification and Change timestamps of the upper file are set to the same value, while the Access timestamp was always set to a value a little earlier. For lower file systems, the behavior of the initial timestamps depended on the volume mode. For a replicated volume, all timestamps were set to same value, while a dispersed volume resulted in different Change and Modified timestamps. Furthermore and as expected, the timestamps across the lower file systems stored on multiple servers varied. Additionally, the Birth timestamp was utilized by the lower file system, but also not populated to the upper file system.

After the modification of a file, the Modification and Change timestamps of the upper file were updated to the same values. Furthermore, the Modification, Change and Access of the lower files were updated.

Since the Access timestamp of an upper file has to be propagated to each lower file, GlusterFS doesn't by default keep track of Access times preventing any performance drops. It was however observed, that access to an upper file could update the Access timestamp of a corresponding lower file depending naturally on the `atime` configuration of the lower file systems. In a setup with three replicated bricks, the specific accessed lower file alternated for each access. Furthermore, the timestamp was not updated for each access, most likely due to again some kind of caching performed within the client. Caching within the GlusterFS servers itself was not observed.

6.1.4. Update behavior for eCryptfs

When a file is created, the Access, Modification and Change timestamps in eCryptfs are all set to the same value, which is the moment the file was written and thus created. The exact same timestamps can be found on the corresponding lower file system. Though it is populated, eCryptfs as well does not utilize or return the Birth timestamp stored in the lower file system.

After the modification of a file, the Modification and Change timestamps were updated and contained the same values within eCryptfs as well as the lower file system. The same holds true for a file access and metadata modification, updating the corresponding Access and Modification timestamps respectively. Consequently, all of the timestamp modifications performed directly on the lower file system were also mirrored to the stacked file system.

6.2. *Key takeaways*

For stacked file system analysis, we advise practitioners to harness both potential sources of timestamps within the upper and lower file system. Extracting timestamps from the upper file system is crucial, as outlined in our previous section for MooseFS and GlusterFS. In addition, timestamps of the lower files should also be extracted and analytical methods need to be able to correlate both timestamp sources. This approach is particularly beneficial in distributed stacked file systems, where data fragmentation leads to a more detailed level of timestamp granularity for each file.

Furthermore, in situations where the upper file system depends solely on the lower file system's timestamps, two aspects should be considered: First, analyzing the lower file system can already provide valuable temporal insights. Second, as our eCryptfs example shows, these timestamps may be more susceptible to manipulation.

Moreover, akin to conventional file system forensics, understanding the behavior of timestamp updates in both the upper and lower file systems is essential.

7. Slack

In traditional file systems, whenever a file's size does not align with the end of a data unit, some unused space between the file's end and the data unit is created. This *file slack*, if not overwritten properly, can contain artefacts of previously stored data within this data unit or can simply be used to intentionally hide data. The exploration of slack space, including its possibilities, detection, and analysis, has already been conducted across various file systems, including NTFS (Huebner et al., 2006), BTRFS (Wani et al., 2020) or APFS (Göbel et al., 2019).

Yet, this scrutiny has not been extended to stacked file systems, which uniquely store a file's content in other files rather than in traditional data units. These lower files may be aligned with a certain *extent* size, e.g. always being a multiple of 4 KiB, resulting in slack space similar to the previously described file slack in traditional file systems. We refer to this slack space as *lower file slack* since it occurs between the actual end of the file and the end of the lower file. Additionally and unlike in traditional file systems, data can be directly appended to a certain lower file directly, since it is represented as a file itself. This way it may be possible to hide data, which is not considered by the upper file system. We define this type of slack as *extra lower file slack*. Fig. 6 illustrates these different types of slack. It is important to understand, if these types of slack exist within a stacked file system and how they can be detected and extracted. During the following experiments, we have evaluated the feasibility of slack space within stacked file systems by utilizing it to hide data.

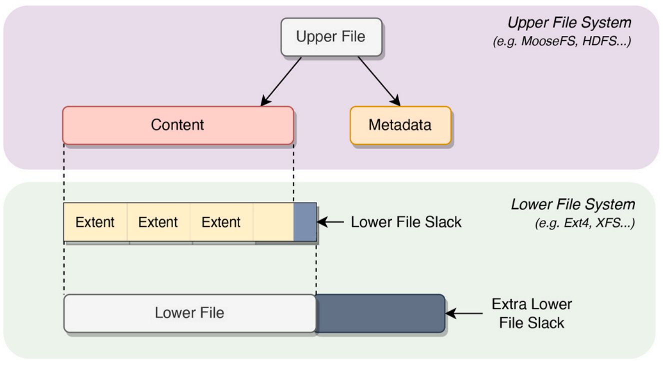


Fig. 6. Overview of possibilities for slack within stacked file systems.

7.1. *Findings*

This section is divided into two parts: the first presents the findings related to the lower file slack space, while the second focuses on the extra lower slack space resulting from expanding the size of a lower file.

7.1.1. *Lower file slack*

7.1.1.1. MooseFS. Chunks start with a `0x2000` byte header, followed by the upper file's content in `0x10000` byte blocks. The final `0x1000` bytes of the chunk header store CRC checksums: four bytes for each block, accommodating up to 1024 blocks. This results in the maximum chunk size of 64 MiB, plus the header size. Given the large block size, MooseFS's lower file slack can be used to hide up to 64 KiB of data without altering the chunk size. Data hidden here doesn't affect the upper file's accessibility or its displayed size. However, inserting data causes a mismatch of the CRC checksums, which led to the chunk marked as INVALID upon server restarts during our experiments. For effective concealment, it's crucial to update these checksums. Furthermore, modifying the upper file doesn't affect the data hidden in the lower file slack. However, if the file expands, reducing the chunk's slack, the concealed data is overwritten.

7.1.1.2. GlusterFS. In distributed and replicated mode, GlusterFS does not utilize any padding and thus the stored lower files are of the exact same size as the corresponding upper files. For this reason, there is no available slack space that can be exploited for data hiding. In dispersed mode, it was observed that the size of a resulting lower file is always a multiple of 512 bytes, theoretically creating slack space that could be used to hide data. However, due to the implemented Erasure coding algorithm, the position and amount of the padding that can be used to reliably hide data varies. Hiding data in the wrong part could lead to modified data within the upper file sometimes even displaying the hidden data in our experiments.

7.1.1.3. eCryptfs. For eCryptfs, the minimum file size of a file stored on the lower file system was always 12 KiB, which includes the 8 KiB header. Its size is then increased in steps of 4 KiB, as this is the *default extent size* used by eCryptfs. The actual extent size can also be found within the header at the start of the lower file. If the data size is not a multiple of the extent size, padding is used and also encrypted. When adding data to this lower file slack, it is still possible to mount eCryptfs and access the file without any problems. For the default extent size, this results in roughly 4 KiB of lower slack space that can be used to hide data. However, as soon as the upper file is modified, the whole contents of the padding is rewritten and the hidden data is lost.

7.1.2. *Extra lower file slack*

7.1.2.1. MooseFS. With stacked file systems, data can also be hidden in the extra lower file slack by appending it to an existing lower file. Since MooseFS utilizes a maximum size for its lower files, it is also ensured that storing data past this offset is protected from being overwritten due to any modifications of the upper file. In our experiments, we filled the space up to the maximum size with zeros and placed data in the succeeding space. Subsequent modifications to the upper file did not overwrite the hidden data in the extra lower file slack. However, as soon as a chunk was transferred to another chunk server, the hidden data did not persist and was lost.

7.1.2.2. GlusterFS. For GlusterFS, hiding data in extra lower file slack proved impractical across all modes. In distributed and replicated modes, data added to the lower file also appears when reading the corresponding upper file, though the upper file size remains unchanged. For replicated volumes with only one replica containing hidden slack data, the upper file consistently reveals this data. When multiple replicas have extra slack data, the upper file reads from the largest lower file. To hide data, one might place it in any replica, but ensure another copy has more benign data, like null bytes. In dispersed mode, data concealed in the slack of one file in a three-brick setup vanished upon reading the upper file, while adding data to two lower files caused an I/O error.

7.1.2.3. eCryptfs. In eCryptfs, an upper file remained accessible with its file size unchanged when the data was stored in the corresponding extra lower file slack. Notably, this appended hidden data persisted when the upper file was modified or when new data was added, provided it did not surpass the padding limit. If the file grew beyond the available padding, the appended data was overwritten. To avoid this, one can add ample padding before the hidden data, ensuring that any growth of the original file only replaces this 'dummy' padding, thereby preserving the hidden data.

7.2. Key takeaways

Stacked file systems differ from traditional ones in that slack space does not contain remnants of previous files, primarily because new lower files are created for each new upper file. However, our findings suggest that exploiting slack space in lower files, or even in additional lower file slack, could be a viable tactic in certain stacked file system implementations. Consequently, forensic practitioners should not only focus on the upper file system but also thoroughly examine the lower file system during their analyses.

Detecting file slack requires a detailed comparison between the file sizes recorded in the upper file system and those of the corresponding lower files. Additionally, cross-referencing replicas of lower files across various lower file systems is critical to identify any discrepancies that may indicate tampering or manipulation.

8. Possibilities for file recovery

Besides operating system or application specific concepts such as trash bins, file deletion is completely file system specific. Some file systems such as older versions of Ext may keep references to the actual data blocks, while others may wipe these entirely. In these experiments, we circumvented the operating system's Trash bin by directly deleting files from the stacked file system using the `rm` command. This approach allowed us to examine the file deletion processes of the stacked file systems in question, thereby identifying the potential methods available for file recovery.

8.1. Findings

8.1.1. MooseFS

MooseFS's own trash mechanism holds deleted files for 24 h by default. When an upper file is deleted, it becomes inaccessible, but its chunks in the lower system persist. These deleted files are labeled as *trash files* in the `NODE` metadata section on the Master Server. Furthermore, the Change timestamp of these deleted upper files stored in the metadata can be used to infer the time of deletion. Notably, even with a trash duration set to zero, chunks stayed active for a couple of minutes. During this time the upper files got tagged as *sustained files* in the metadata indicating they were deleted but still open. The Change timestamp of these files can hint at their deletion time. Once a file was fully deleted, its chunks were too.

8.1.2. GlusterFS

In its default configuration, GlusterFS does not utilize its *Trash translator* feature. Thus, as soon as an upper file is deleted, the corresponding files in the lower file system are removed as well and the possibilities of recovery depend on the lower file system. Enabling this feature results in the creation of a `.trashcan` directory on each of the bricks, which is used to hold deleted upper files and is also mounted within the upper file system. After the deletion, the GFID-named lower file remains intact, while the hard link in the original hierarchy is removed from the lower file system. Instead, a new hard link within the `.trashcan` directory is created, whose name consists of the original upper file's name and the actual time of deletion. Furthermore, the original path hierarchy of the deleted file is also recreated within the trash directory.

8.1.3. eCryptfs

Following a file deletion within eCryptfs, the corresponding lower file was also deleted instantaneously in our experiments.

8.2. Key takeaways

Our research reveals that stacked file systems can offer an extra opportunity for file recovery through their own trash features. Understanding the specific structure and metadata of the stacked file system is key, and the data from these trash bin mechanisms should be factored into the analysis process.

Investigators should consider the new opportunities presented by the presence of an additional lower file system. Even if content is deleted from the upper file system, the original data might still exist as lower files within the lower file system. While file recovery becomes wholly dependent on the lower file system following a complete file deletion, the inherent structure of these lower files can be utilized for advanced recovery techniques, such as file carving. In summary, these findings imply that acquiring the lower file system, either physically or logically, is more advantageous than merely performing a simple logical acquisition of the upper file system.

9. Conclusion

Contrary to traditional file systems, the concept of stacked file systems utilizes an additional file system for data storage. Given its integration into various modern distributed file systems, encountering stacked file systems is inevitable in present and future forensic investigations. In this paper, we focused on the forensic analysis of stacked file systems and presented an updated model that is capable of handling this class of file systems. Complementing this, we presented various forensic implications based on traditional analysis techniques and explored them using three representative stacked file systems as examples.

Our findings reveal that understanding the architecture, mechanisms, and features of stacked file systems is crucial for effective analysis. We demonstrated basic procedures like identification and metadata extraction in our findings, noting that further research is essential for a more comprehensive understanding of these systems. The significance of the underlying file system was also emphasized, particularly its potential to enhance investigations with finer details, such as more precise timestamps. Notably, even when access to the upper file system itself is hindered, for example by encryption or incomplete distributed structures, valuable data can still be retrieved from the lower file system.

To fully leverage these insights, it is imperative for current forensic methodologies and tools to adapt. Our research lays a solid groundwork for future exploration in this area and aims to increase awareness among forensic investigators regarding the complexities and opportunities presented by stacked file systems.

Acknowledgement

We thank our shepherd and our anonymous reviewers for their invaluable feedback on this paper.

References

Aranya, A., Wright, C.P., Zadok, E., 2004. Tracefs: a file system to trace them all. FAST 129–145.
Asim, M., McKinnel, D.R., Dehghantanha, A., Parizi, R.M., Hammoudeh, M., Epiphaniou, G., 2019. Big data forensics: Hadoop distributed file systems as a case study. Handbook of Big Data and IoT Security 179–210.
Bhat, W., Quadri, S., 2012. Restfs: secure data deletion using reliable & efficient stackable file system. In: 2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (Sami). IEEE, pp. 457–462.
Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Professional.
Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12.
Göbel, T., Türr, J., Baier, H., 2019. Revisiting data hiding techniques for apple file system. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, vols. 1–10.
Halcrow, M.A., ecryptfs, 2005. An enterprise-class encrypted filesystem for linux. Proceedings of the 2005 Linux Symposium 1, 201–218.
Harshany, E., Benton, R., Bourrie, D., Glisson, W., 2020. Big data forensics: Hadoop 3.2.0 reconstruction. Forensic Sci. Int.: Digit. Invest. 32, 300909.
Heidemann, J.S., Popek, G.J., 1994. File-system development with stackable layers. ACM Trans. Comput. Syst. 12 (1), 58–89.
Hilgert, J.N., Lambertz, M., Plohmann, D., 2017. Extending the sleuth kit and its underlying model for pooled storage file system forensic analysis. Digit. Invest. 22, S76–S85.
Huebner, E., Bem, D., Wee, C.K., 2006. Data hiding in the ntfs file system. Digit. Invest. 3 (4), 211–226.
Martini, B., Choo, K.K.R., 2014. Distributed filesystem forensics: Xtreemfs as a case study. Digit. Invest. 11 (4), 295–313.
Raghavan, S., 2013. Digital forensic research: current state of the art. Csi Transactions on ICT 1, 91–114.
Sipek, J., Pericleous, Y., Zadok, E., 2007. Kernel support for stackable file systems. In: Proc. Of the 2007 Ottawa Linux Symposium, vol. 2. Citeseer, pp. 223–227.
van der Meer, V., Jonker, H., van den Bos, J., 2021. A contemporary investigation of NTFS file fragmentation. Forensic Sci. Int.: Digit. Invest. 38, 301125.
Wani, M.A., Bhat, W.A., Dehghantanha, A., 2020. An analysis of anti-forensic capabilities of b-tree file system (btrfs). Aust. J. Forensic Sci. 52 (4), 371–386.
Zadok, E., 1999. Stackable File Systems as a Security Tool. Tech. Rep.; Citeseer.